# Autonomous Drone Maze Traversal and Image Detection

Prepared by

💯Percent Throttle
6846 BeagleBone Drive, Ann Arbor, MI
746-794-9845

Elizabeth Lu, Ethan Perlmutter, Vidya Srinivas, Jason Williams
Engineers in Training

Project dates: September 3, 2019 - December 10, 2019

Prepared for
Dr. Robert Dick, Instructor, University of Michigan
Dr. Sarah Burcon, Instructor, University of Michigan

December 10, 2019

# Table of Contents

**List of Figures**……………………………………………...……………………………..…………..iii

**Executive Summary**……………………..…………………………………...………………………....iv

**Introduction and Background**…………….………………...…………………...……………....…..…..1

*Overview of Prototype and Subsystems*…………………………..……….…………………….1

*Structural and Mechanical Components*………………...…………..………………………….1

*Sensor Details*……………………………………………………….….……………………….2

*Overview of Roll, Pitch, and Yaw, and Throttle*……………….………………….……...…..........3

**Overview of PID Control**…………………...…………………………………………….….….…….4

*The Consequences of Changing Individual Components of the PID Controller*……….…………4

*The Importance of PID Control For Desired Behavior at Takeoff*…………………...……...…5

**Overview of Device Function and Code**……....……………………………………….……………....5

*Initial Setup and Testing of Quadcopter*……………………………………...………………5

*Design Constraints and Specifications*……………………...…………………….……………6

*Maze Navigation Technique*……………………………………….……………….…………6

*Maze Navigation Testing*…………………..…………………....…………….………..…7

*Code Structure and Function*…………………………...…………………….……………7

*Challenges Encountered*.……………………………………………………...……...…………7

*Maze Navigation Results*…………..………………………………………………….……….8

**Overview of Custom Design Project** …….………………………………...……………...……......…8

*Initial Setup of PixyCam*.…….……………………………………………………….…...……....8

*Hardware Components and Connections*.....……………………………………………….......9

*Code Structure and Function*…….……………….……………………....………….........…10

*Challenges Encountered*…………………………………………………………………….10

*Custom Project Results*…….………………………………………………………......…....11

*Custom Project Future Applications*.…….……………………….……….……………….......11

**Recommendations For Further Development**………………...……………………………….……12

**Conclusion**……………………...…………………..…………………………………….………12

**References**…………………………………………………...…………………...…………...…… 14

**Appendix A: Maze Navigation Code**………………………………………………………...….15

**Appendix B: Pixy Cam Code**…………………………....……………………………………25

**Appendix C: Technical Specifications**…………………….…………………………………….31

# Figures and Tables

## Tables

1. Variables and Descriptions……………………………………………………..…………………5
2. Base PID Control Parameters…………………………………………………………………...6

## Figures

1. Propeller Configuration and Direction and Propeller Shape………………………………………2
2. Quadcopter Sensors and Components…………………………………………………...…...2
3. Roll, Pitch, Yaw, and Throttle Implementations…………………………………………...…...3
4. Consequences of Changing Individual Components of the PID Controller……………………….4
5. Maze Navigation Technique and PID Parameters…………………………………………………6
6. Maze Navigation Code Flowchart……………………………………………………………....7
7. PixyCam, Signature 1, Orange………………………………………………………………......9
8. Voltage Divider Schematic and Physical Implementation……………………....….…………...9
9. UART1 Serial Port Configuration……………………………………………………….……...9
10. Modified LIDAR Code for PixyCam…...…………………………….…………………….....10

# Executive Summary

On February 23, 2019, Prof. Robert Dick, President, was contacted by Ariadne de Bothezat, a client who is planning to develop and market quadcopter platforms and sensor-electronic kits to enthusiasts and universities. We were asked by Prof. Dick to prepare a formal report about our quadcopter performance and findings in our Engineering 100 class. The final deliverables for this class were an algorithm for the autonomous flight and navigation of a quadcopter through a given maze, and a custom design innovation on the existing quadcopter system.

Design constraints and specifications were that the quadcopter was required to fly through the given maze from start to end without hitting any walls or other obstacles, as fast as possible. The autonomous maze navigation algorithm was required to be reliable in the presence of adverse conditions and was required to incorporate infrared (IR)-sensor filtering and Proportional Integral Derivative (PID) controller tuning, resulting in minimal collisions during flight. Teams were also expected to propose an additional feature for the existing quadcopter configuration, that either enhanced quadcopter performance in the given maze, or innovated on the existing quadcopter design through the incorporation of a new sensor or actuator.

The quadcopters discussed in this report incorporate both software and hardware components. The quadcopter subsystems consisted of multiple onboard sensors, such as proximity and orientation sensors. These allowed the quadcopter to detect and react to its surroundings. PID control signals were sent to the quadcopter to specify certain behavior based on different environmental conditions.

We were successful in our navigation of the maze, completing it in about 25 seconds with minimal collisions. Our main maze navigation algorithm was written in the C++ programming language. This code was uploaded to the quadcopter's main control system, which then routed control signals to the quadcopter. We controlled the quadcopter through Mission Planner, a ground control system for ArduPilot, which is a control software for drones and other autonomous systems. All code was written within a set autonomous controller, which was then modified through the use of code structures that incorporated selection.

Our custom design innovation is an autonomous landing algorithm based on color detection. This algorithm is intended to have real-world applications in the area of unmanned aerial vehicle safety. We completed the hardware setup of a camera for color detection, and also connected the camera to the quadcopter. Our goal was to mount the PixyCam setup on the quadcopter, fly it through the maze, and land whenever the quadcopter whenever it flew over a specific color. However, though we accomplished all other parts of this goal, we were unable to fly the quadcopter with the PixyCam setup due to time constraints and fragile hardware connections. Lastly, further recommendations for the application of the maze navigation project and custom project are also discussed in this report.

The primary topics discussed in this report include an overview of PID controllers and control systems, an overview of the subsystems of the quadcopter used for autonomous navigation of the maze, and base technical specifications for quadcopter flight. Primary topics also include design constraints and specifications for maze navigation, function of the implemented algorithm, overview of the written code, documentation of the custom design project, and recommendations for further development related to the project.

# Introduction and Background

On February 23, 2019, Prof. Robert Dick, President, was contacted by Ariadne de Bothezat, a client who is planning to develop and market quadcopter platforms and sensor-electronic kits to enthusiasts and universities. Prof. Dick asked us to prepare a formal report detailing our quadcopter performance and general findings in our Engineering 100 class. The final deliverables for this class were an algorithm for the autonomous flight and navigation of a quadcopter through a given maze as well as a custom design innovation on the existing quadcopter system.

The primary topics discussed in this report include an overview of PID controllers and control systems, an overview of the subsystems of the quadcopter used for autonomous navigation of the maze, and base technical specifications for quadcopter flight. Primary topics also include design constraints and specifications for maze navigation, function of the implemented algorithm, overview of the written code, documentation of the custom design project, and recommendations for further development related to the project.

**Overview of Prototype and Subsystems**

A quadcopter uses many interrelated components in order to operate smoothly and efficiently. These components include different types of hardware such as structural components, mechanical components, and sensors. Structural components allow all parts of a quadcopter to be connected securely, while mechanical components allow parts of a quadcopter to move. Lastly, sensors are used to detect changes in a quadcopter's environment. The following section contains details on the components discussed above.

**Structural and Mechanical Components**

A quadcopter is an aerial vehicle that has four rotors and four arms. The quadcopter chassis, or frame, is a structural framework that holds all components of a quadcopter in place while it flies. We used an X-shaped frame along with a BeagleBone Blue microcontroller, Arduino Uno board, and batteries, all of which were housed in the frame's central intersection. Two types of lithium polymer batteries were used: a two-cell battery (9V) used to power the BeagleBone Blue and a three-cell battery (12V) used to power the motors. Additionally, four electronic speed controllers were positioned on top of each of the arms of the frame, and four motors and their respective propellers were on the end of each arm.

A quadcopter has two counterclockwise (CCW) propellers located across from each other, and two clockwise (CW) propellers, also located across from each other. One difference between CW and CCW propellers is that they are shaped differently. This is to ensure that when a propeller is rotating in its intended direction, the side of the propeller with a consistent curve is be the leading edge, while the other thinner side is the trailing edge. Another difference between the CCW and CW propellers is the direction of the thrust produced when the motors spin. For example, when a CCW propeller spins CCW, it produces upward thrust, and when it spins CW, it produces downward thrust. On the other hand, when a CW propeller spins CCW, it produces downward thrust, and when it spins CW, it produces upward thrust. When quadcopter propellers spin in their intended directions, they always produce upward thrust. Figure 1 (on page 2) shows a diagram of the direction of propeller rotation. In addition, Figure 2 (on page 2) displays the components of a quadcopter, with magnified views of major components.

Figure 1: Propeller Configuration and Direction (Left) and Propeller Shape (Right)

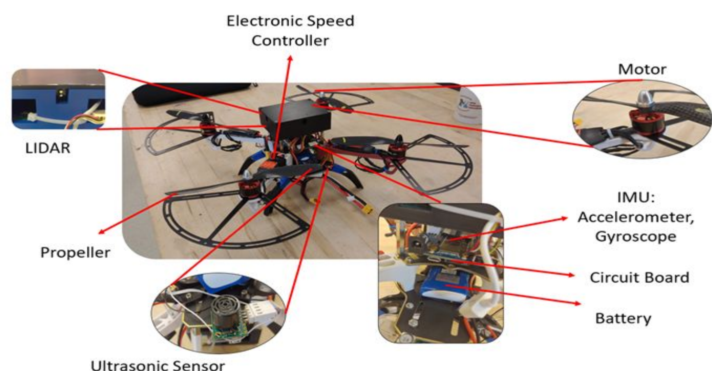



Figure 2: Quadcopter Sensors and Components

**Sensor Details**

An autonomous drone uses many sensors to gather information from its surroundings in order to deduce where it is located in space. These sensors include an accelerometer,  a gyroscope, magnetometers, an ultrasonic sensor, and Light Detection and Ranging (LIDAR). Figure 2 below displays these sensors.

*Accelerometer.* A quadcopter uses three accelerometers to measure acceleration along three axes. It always measures the force of gravity, and also can detect movement and vibration. It is a type of Micro-Electro-Mechanical Sensor (MEMS) that functions by detecting the displacement of a small mass suspended by support beams in an integrated circuit.

*Gyroscope.* A gyroscope measures change in rotational angle per unit time. It functions by detecting the angular displacement of a mounted rotating disk on a spinning axis from a more stable, larger wheel. This sensor allows a quadcopter to maintain its stability.

*Ultrasonic Sensor.* An ultrasonic sensor measures the altitude of a quadcopter. It sends a continuous train of 40 kHz square sound pulses of 1 ms followed by 9 ms of silence. It then detects how long it takes for the sent signal to reflect off the ground and back to the sensor, using the speed of sound to determine altitude.

*LIDAR.* The drone uses four LIDAR sensors, one on each side, to measure horizontal distance and four directions. LIDAR sends out a light signal similar to the sound signal sent out by the ultrasonic sensor, and also uses a similar method to determine distance.

**Overview of Roll, Pitch, Yaw, and Throttle**

The terms roll, pitch, and yaw describe quadcopter motion along each of the main axes. The term throttle refers to the speed of each motor (higher throttle corresponds to higher speed). A quadcopter is able to control aspects of its motion by utilizing various motor speeds. Figure 3 below shows a diagram of roll, pitch, yaw, and throttle for a quadcopter.

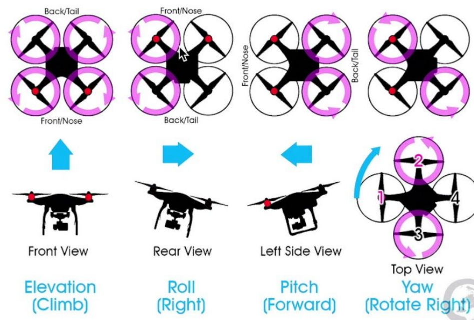



Figure 3: Roll, Pitch, Yaw, and Throttle Implementations [2]

*Roll.* Roll is motion to the left or right. When a quadcopter rolls, either its left or right side is tilted downwards while its other side it tilted upwards. This is achieved by decreasing the speed of a pair of side motors while increasing the speed of the opposite pair.

*Pitch.* Pitch is motion forward or backward. When a quadcopter pitches, either its front or back side is tilted downwards its other side is tilted upwards. This is achieved by decreasing the speed of the front pair of motors while increasing the speed of the back pair or increasing the speed of the back pair while decreasing the speed of the front pair.

*Yaw.* Yaw is rotational motion about a vertical axis. A quadcopter yaws by increasing the speed of a pair of its diagonal motors while decreasing the speed of the other two.

*Throttle.* Throttle, or motion along the vertical axis, allows a quadcopter to increase or decrease its altitude. Increasing the speed of all four motors concurrently will cause a quadcopter to rise, while decreasing the speeds of the motors concurrently will cause it to descend.

The main board (BeagleBone Blue) is responsible for setting the speeds of the motors based on algorithms and sensor readings. It sends pulse-width-modulated (PWM) signals to the electronic speed controllers on the quadcopter arms, which then convert these signals and send them to the motors, changing motor speeds appropriately.

# Overview of PID Control

A Proportional Integral Derivative Controller, or PID controller, is a closed loop control system. A closed loop control system is a control system that regulates itself without any human input. The purpose of a PID controller is to get a system from current value (process variable) to a desired value (setpoint value). Based on changes to the proportional, integral, and derivative terms, a system can demonstrate different behavior. The effects of these changes will be investigated in the next section of this report.

**The Consequences of Changing Individual Components of the PID Controller**

*The Proportional Term (P).* The proportional term is directly proportional to the error term in the system. The error term is calculated by taking the difference between the system's current value and the system's target value.When tuned properly, the proportional term will allow a system to smoothly and quickly come to steady state. However, increasing the proportional term by too much can cause oscillations in the system. Thus, we use the proportional term with the integral and derivative terms. Figure 4 below shows various values of the proportional term and a system’s behavior for each value. The target value of the system is also shown for comparison.

*The Integral Term (I).* The integral term is a cumulative error term, meaning that it sums error over time. Once error in a system is eliminated, this term does not grow anymore. However, because the integral term only takes into account past error when used by itself, it can make system response sluggish. Therefore, this term is usually used with the proportional term and the derivative term. When used with these terms, the integral term  can reduce or eliminate steady-state error. Figure 4 below shows various values of the integral term and a system’s behavior for each value. The target value of the system is also shown for comparison.

*The Derivative Term (D).* The derivative term is a predictive term which attempts to cancel out errors that will occur in the system in the near future. This term is very sensitive to noise in the system.  Therefore, the derivative term is used with the proportional and integral terms to counter its sensitivity. Figure 4 below shows various values of the derivative term and a system’s behavior for each value. The target value of the system is also shown for comparison.

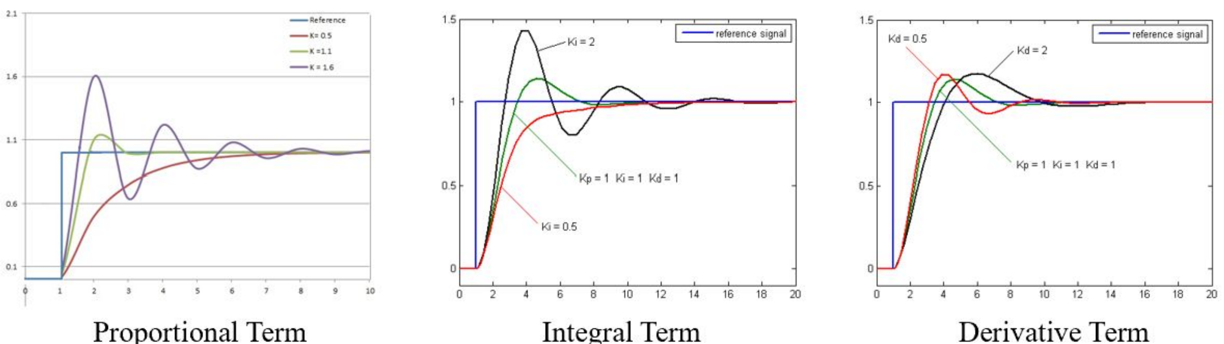



Figure 4: Consequences of Changing Individual Components of the PID Controller [3]

### The Importance of PID Control For Desired Behavior at Takeoff

To achieve smooth and quick takeoff, one can tune the proportional gains of a PID controller. The proportional gains, $K_{p1}$ and $K_{p2}$, are tuned differently according to desired takeoff behavior, but in order to maintain a certain behavior, the ratio between the proportional gains must be maintained. Once an ideal ratio between the proportional gains is achieved, the quadcopter will achieve a smooth takeoff.

# Overview of Device Function and Code

The following section contains information about quadcopter testing before and during maze navigation, design constraints and specifications, maze navigation code and technique, and challenges encountered during maze navigation.

### Initial Setup and Testing of Quadcopter

Initial setup of our quadcopter included flying while tethered to a weight to constrain the quadcopter's fly area and prevent any unexpected actions from occurring . We then used the altitude hold (AltHold) fly mode in ArduPilot, which brought the quadcopter up to a specified target altitude, without constraints on yaw, roll, or pitch. The LIDAR proximity sensors were not used in the AltHold fly mode. After tuning the proportional gains for our quadcopter to achieve smooth takeoff, we began to program the quadcopter to avoid obstacles using built-in obstacle avoidance parameters in Mission Planner. Lastly, we were given the ArduCopter source code and a C++ file, `control_autonomous.cpp`, which contained functions that allowed for autonomous navigation of the quadcopter.

The function that we used within the given C++ file was called `autonomous_controller`, a function that returned a boolean value, with a return value of true indicating that the quadcopter should continue to fly, and a return value of false indicating that the quadcopter should land. This function was executed at a frequency of 400 Hz, or 400 times per second, during flight. Within this function, we were able to access and set proximity sensor (LIDAR) readings, altitude sensor (sonar) readings, and target pitch, roll, and yaw rate variables. Descriptions of these variables are located in Table 1 below. Using this function, we wrote a simple algorithm to center our quadcopter between two walls, and tuned our base PID control parameters (proportional, integral, and derivative terms) to ensure steady behavior of the quadcopter. These values appear in Table 2 (page 6) below.

| Variable | Type | Description |
|---|---|---|
| dist_left | float | Distance to quadcopter's left in decimeters (LIDAR) |
| dist_right | float | Distance to quadcopter's right in decimeters (LIDAR) |
| dist_forward | float | Distance from quadcopter's front in decimeters |
| dist_backward | float | Distance from quadcopter's back in decimeters (LIDAR) |
| rangefinder_alt | float | Quadcopter's vertical altitude in meters |
| step | static int | Indicates current navigation step through the maze (1-6) |
| target_roll | static float | Indicates the quadcopter's current roll |
| target_pitch | static float | Indicates the quadcopter's current pitch |

Table 1: Variables and Descriptions

| | Pitch | Roll |
|---|---|---|
| Proportional | 1.00 | 1.00 |
| Integral | 0.50 | 0.25 |
| Derivative | 1.70 | 1.00 |

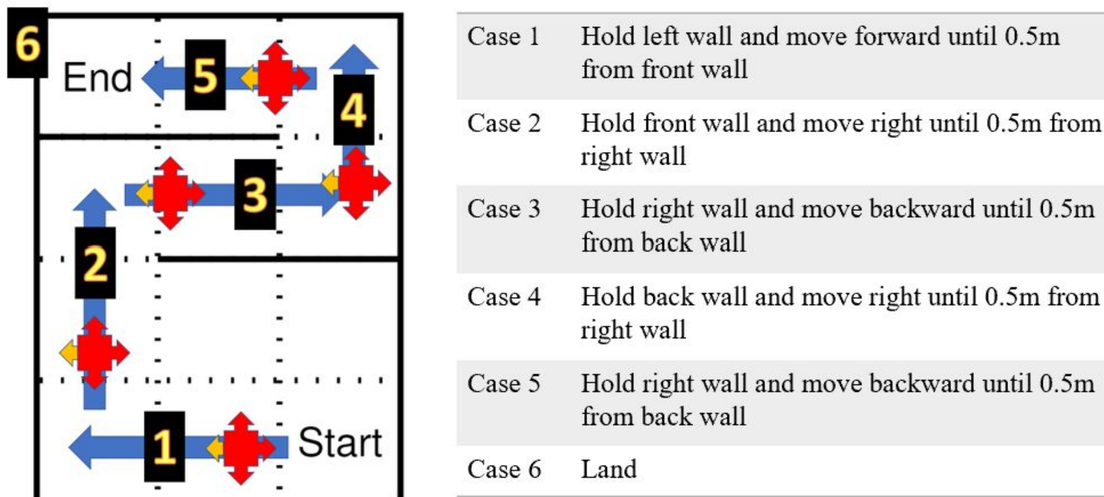
Table 2: Base PID Control Parameters

**Design Constraints and Specifications**

Design constraints and specifications for the maze were that the quadcopter had to fly through the maze from start to end without hitting any walls or other obstacles, as fast as possible. The autonomous maze navigation algorithm was required to be reliable in the presence of adverse conditions and was required to incorporate infrared (IR)-sensor filtering and PID controller tuning, resulting in minimal collisions during flight. We were also expected to propose an additional feature for the existing quadcopter configuration, that either enhanced quadcopter performance in the given maze, or innovated on the existing quadcopter design through the incorporation of a new sensor or actuator.

**Maze Navigation Technique**

To navigate the maze, we first planned a path for the quadcopter to take through the maze, as indicated by the arrows in Figure 6 below. This path was determined to be the most efficient path for the quadcopter to take through the maze, as it incorporated the least number of turns, and also allowed the quadcopter enough distance over which to read from its proximity sensors. Next, the maze was divided into six distinct steps, also shown in Figure 6 below. In the first five steps, the quadcopter moves through the maze, while in the last step, the quadcopter lands. The first five steps of the maze were executed in a similar manner. For each of these steps, two PID controllers were used: one that held the quadcopter at a distance from one wall, and one that moved the quadcopter in a certain direction until it detected a wall in its path. For example, in step one in Figure 6 below, the quadcopter holds to the left wall and also moves forward. The table in Figure 6 below displays the specific directions and distances for the PID controllers used for each step indicated. The yellow arrow on this figure indicates the front of the quadcopter.




| | |
|---|---|
| Case 1 | Hold left wall and move forward until 0.5m from front wall |
| Case 2 | Hold front wall and move right until 0.5m from right wall |
| Case 3 | Hold right wall and move backward until 0.5m from back wall |
| Case 4 | Hold back wall and move right until 0.5m from right wall |
| Case 5 | Hold right wall and move backward until 0.5m from back wall |
| Case 6 | Land |

Figure 5: Maze Navigation Technique (Left) and PID Parameters (Right)

## Maze Navigation Testing

Before and during test runs through the maze, we tested both software and hardware components of our quadcopter setup. We began our testing using a three-panel board. With our drone attached to a tether, we repositioned the panels to mimic maze walls. We then flew our drone to see if it moved in the intended direction. Next, at the maze test site, we removed the propellers from our drone. With propellers removed, we started the drone and simulated its flight by lifting it off the ground and moving it. We placed obstacles in front of the LIDAR sensors and checked both our sensor readings and code steps to ensure that they were reading as intended. Lastly, before flying the quadcopter through the maze, we walked the quadcopter through the maze as a final check before flight.

## Code Structure and Function

Our main maze navigation algorithm was written in the C++ programming language. This code was uploaded to the quadcopter's main control system, which then routed control signals to the quadcopter, enabling it to behave as the code intended. Within the `autonomous_controller` function, we wrote a state machine code structure, also known as a switch-case in C++. In the context of maze navigation the "case" in the switch-case is given by the step number (see Figure 5 above). Within each case specified by a distinct step, we have PID controllers that keep the quadcopter a certain distance from one wall at all times, and also move the quadcopter in the intended direction. At the end of each of the first five steps, there is a conditional statement (`if`) which checks if the quadcopter is a certain distance from a wall and changes the step accordingly. Lastly, at the end of the function, the code returns either a true or false value. A true return value allows the quadcopter to keep flying, while a false return value causes the quadcopter to land. Figure 6 below shows a flowchart of the code that was implemented inside the `autonomous_controller` function with conditional statements in yellow diamonds, and return values in red circles (see Appendix A for more details about maze navigation code).

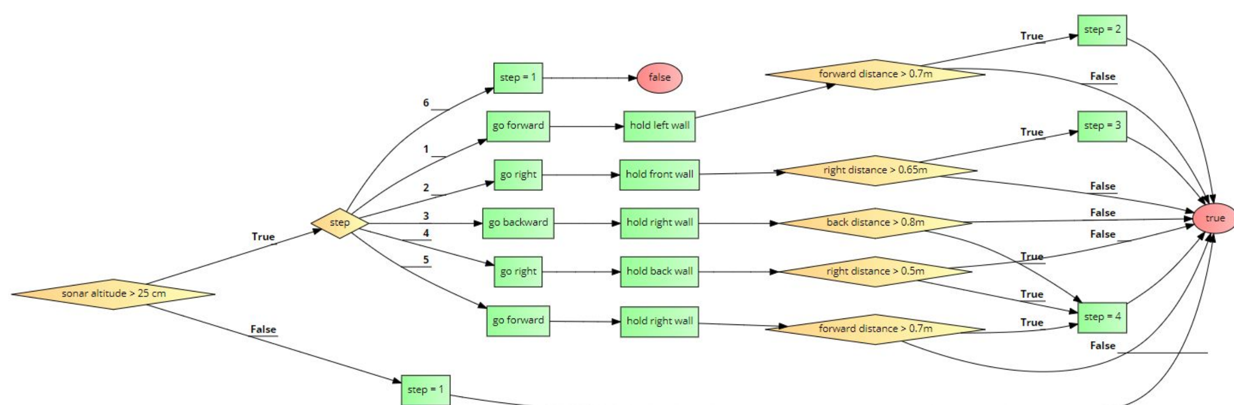



Figure 6: Maze Navigation Code Flowchart

## Challenges Encountered

We encountered both hardware and software challenges during test flights in the maze.

*Hardware Challenges.* One of our primary challenges was that our LIDAR sensors froze every time our quadcopter went into the air. We determined that this problem was a result of interference between continuous LIDAR readings and continuous printing of debugging messages from our C++ code. To fix

this, we printed fewer debugging messages. Also, for a short time, our quadcopter's ultrasonic sensor also experienced severe fluctuations in reading. We found that this problem was due to a loose connection between the BeagleBone board and the ultrasonic sensor and soon fixed the problem.

*Code Challenges.* Within our code, we encountered problems with variable scope. Specifically, when testing in the maze, we realized that the target pitch and roll values that we were sending to the quadcopter were not being read. This was due to a variable scope problem, in which variables defined within the `autonomous_controller` function were initialized every time the function was called (400 times per second). We fixed this problem by using the keyword `static` in C++, which allows a variable to hold its value between function calls. Apart from this problem, we only encountered minor syntax and semantic errors in our code.

*PID Tuning Challenges.* Lastly, we encountered PID parameter tuning challenges during flight in the maze. These challenges were mainly because our quadcopter exhibited different behavior during separate runs due to battery voltages and other unanticipated conditions. Since we were unable to filter out these unpredictable parameters, we recalibrated and reset our quadcopter in between runs to minimize problems during maze navigation. We also ensured that the batteries used were close to full voltage, and recalibrated our PID parameters during every flight session.

### Maze Navigation Results

We were successful in our navigation of the maze, completing it in about 25 seconds in our best run. During our best flight, our quadcopter smoothly and efficiently navigated the maze with minimal collisions. Most of our quadcopter flight trials averaged around 30 seconds. Throughout the testing process, we completed around 35 test flights with propellers attached, 6 maze simulations, and hourly LIDAR and sensor tests and calibration.

For additional details on the challenges encountered and solutions implemented in the maze navigation phase, see Appendix C.

## Overview of Custom Design Project

The goal of our custom design project was to hook up the PixyCam, an Arduino-compatible camera to our drone and train it to recognize a specific color. Upon recognition of this color, we wanted the drone to land on or near the spot of color. This custom project is intended to have real-world applications in the area of unmanned aerial vehicle safety.

### Initial Setup of PixyCam

The PixyCam setup included connecting the PixyCam to PixyMon2, a desktop application that allows one to configure and train the PixyCam, as well as see what it sees. Once the PixyCam was powered via a USB cable connected to a computer, we set a color signature by capturing an image with the PixyCam and choosing our desired color from a section of the image [4]. In our case, we set a color signature of orange (Signature 1) from a sheet of colored paper as seen below in Figure 7 (page 9).

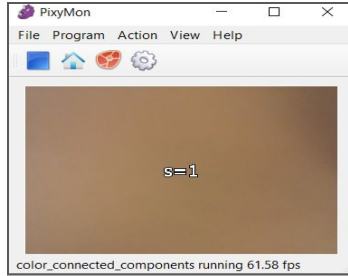



Figure 7: PixyCam, Signature 1, Orange

**Hardware Components and Connections**

To set up a connection between the PixyCam and the BeagleBone, we opened up and configured a second serial port for connection to a second Arduino. We used a Sparkfun Redboard as our second Arduino, which had 5V serial ports. However, the BeagleBone Blue that we were connecting to had 3.3V serial ports. As a result, to avoid damage to the BeagleBone's serial port (which would result from passing too high of a voltage through), we were required to build a voltage divider as seen in Figure 8 below.

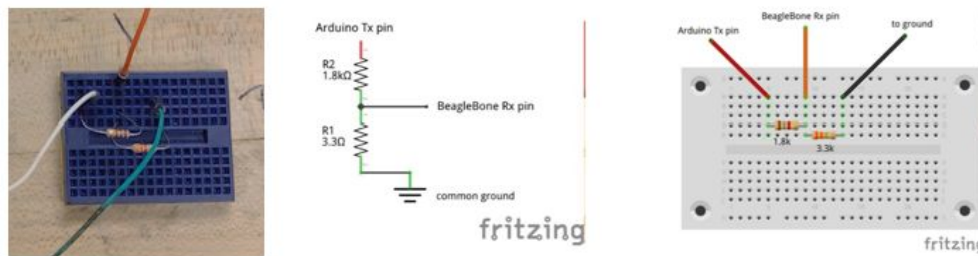



Figure 8: Voltage Divider Schematic and Physical Implementation

The voltage divider was built with a 3.3kΩ resistor and a 1.8kΩ resistor to drop the 5V serial signal to 3.3V. To open connection between the BeagleBone and Arduino, we opened a second Universal Asynchronous Transmitter (UART) serial port as seen in Figure 10 below. This was done by accessing the BeagleBone's root directory through an Ubuntu Virtual Machine (Linux environment) using Secure Shell (SSH) and setting the "Serial 2 Protocol" in Mission Planner to "1" to configure the UART 1 serial port (see Figure 9 below).

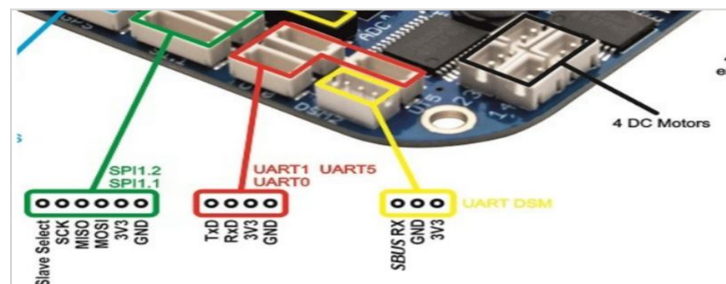



Figure 9: UART1 Serial Port Configuration

**Code Structure and Function**

The code for this custom design project was written in the C++ programming language using the Arduino integrated development environment (IDE). Due to time constraints, we modified existing code that was being used to send distance readings from the LIDAR to the UART5 serial port in order to interface our PixyCam with the BeagleBone Blue. We first included the library `pixy2ccc.h` which contained all native functions that came with the PixyCam. We then made a copy of the LIDAR code and then commented out all initializations of the LIDAR sensors. This was to ensure that the code ran even without physical LIDAR being connected to our second Arduino board. Finally, we wrote a statement that detected whether the PixyCam detected Signature 1 (orange), which we had trained it to recognize before. Inside this statement, using the same function that would be used by the LIDAR to send distance measurements, we set the value of all four LIDAR readings to zero, as seen in Figure 10 below (see Appendix B for details about PixyCam code).

```
if (pixy.ccc.blocks[i].m_signature == 1){
    send_distance( 0, 0 );
    send_distance( 0, 1 );
    send_distance( 0, 2 );
    send_distance( 0, 3 );
  }
```

Figure 10: Modified LIDAR Code for PixyCam

We then went into our C++ code (specifically the `autonomous_controller` function) and wrote a conditional statement to check if the BeagleBone received four LIDAR readings of zero. If this was the case, we returned a value of `false` for the function, therefore causing the quadcopter to land if the color orange was detected. To check if the connections above (both hardware and software) were behaving as intended, we used a Linux environment to SSH into the BeagleBone Blue root directory. From here, we accessed the `dev` file directory, a directory that contains details about devices or other special files. We then opened the `ttyO1` file (corresponding to the UART1 serial port) using the Linux command `cat` which allowed us to see whether any messages were being sent to the BeagleBone from the Arduino. We only attempted to test with the quadcopter motors on after we confirmed the connection between the Arduino, BeagleBone, and PixyCam using this method.

**Challenges Encountered**

We encountered both hardware and software challenges while attempting to connect our PixyCam, Arduino, and BeagleBone Blue.

*PixyCam Reading Challenges.* One of the challenges we encountered during our PixyCam custom project was inconsistent readings from the PixyCam. When the PixyCam reads a set signature, a small LED lights up, signaling that the camera sees the signature. However, during our testing process we noticed that the LED signal was activated even when our signature was not present. This could be due to poor lighting conditions and lack of better color training. Most likely, there was an unforeseen issue in either the software or hardware that caused a faulty communication in which the PixyCam sent a signal every time

the camera saw any signature rather than the signature we set. We were not able to determine where in our system the problem was located.

*Arduino Connection Challenges.* Another challenge we encountered was establishing a connection between the Arduino Sparkfun Redboard, BeagleBone, and PixyCam. We needed to open a second serial port on the BeagleBone and create a voltage divider to avoid damage between the 3.3V BeagleBone board and the 5V Arduino board. Even when the second serial port was open, and we established a connection between the Arduino, PixyCam, and BeagleBone, we had software issues, which resulted in us not properly sending the intended signals to the BeagleBone. Since our code was based on the given Arduino LIDAR code, we encountered issues where the Arduino board expected physical LIDAR sensors to be connected (which was not the case), and therefore would not run. As a result, we had to comment out any code that initialized the LIDAR sensors. In addition, to avoid any further interference with messages sent to the BeagleBone, we took out the PixyCam `Serial.print` messages that initially printed out the signature the PixyCam read.

*Serial Port Pin Challenges.* A smaller issue we came across was uploading our code to the Arduino board. While uploading our code, serial data was coming from our computer while the input/output (I/O) pins (pins 0 and 1 on the Arduino) were affected by connections to the BeagleBone. The Arduino serial data conflicted with the computer's serial data and as a result, the code failed to upload. To solve this issue, we disconnected the two pins (pins 0 and 1), and were then able to upload the sketch as normal. This problem is known to affect certain Arduino boards such as the Uno, Due, and Mega [1].

For additional details on the challenges encountered and solutions implemented in the custom project phase, see Appendix C.

### Custom Project Results

We established a connection between our quadcopter and PixyCam through serial connections, an Arduino RedBoard, and a BeagleBone Blue. Through this connection, in our trial runs, anytime the signature 1 (orange) was recognized, the PixyCam sent a landing signal to the quadcopter. Our goal was to mount the PixyCam setup on the quadcopter, fly it through the maze, and land whenever the quadcopter whenever it flew over a specific color. However, we were unable to finish this last step due to time constraints and fragile hardware connections.

### Custom Project Future Applications

Allowing a quadcopter to use color detection through a camera will allow the quadcopter to identify a place to land. This will enable the quadcopter to identify a safe zone to land in, and will allow for autonomous landing of the quadcopter. Similar techniques can be used to allow other unmanned aerial vehicles to detect safe landing zones. This is also another step toward image recognition for drones. This can be expanded on in the future, in order to enable drones to detect other important obstacles and markings.

# Recommendations for Further Development

Currently, our quadcopter is hard-coded to navigate only the given maze. We use switch-case states specific to each segment of the current maze, and therefore, our code is only valid for one maze. A recommendation for future development would be to fully automate the drone, so that it can navigate any maze or structure. This is possible through the use of the right-hand rule, or another similar technique for maze navigation. Another improvement for the future would to further tune the quadcopter's PID controller parameters to decrease navigation time and contact with walls. With further testing and optimization of the PID control parameters, the drone should be able to react more quickly to its surroundings and complete a maze in less time.

To further develop our color detection with the PixyCam, we recommend incorporating edge detection in addition to color detection, so the quadcopter can identify objects' shapes in addition to their colors. An edge detection algorithm detects sudden changes in brightness or pixel structure, allowing it to detect where a shape or object is. This makes it possible to perform tasks such as image identification and environment mapping. In the quadcopter context, edge detection could allow a quadcopter to map out its surroundings, which would allow it to navigate with a higher degree of autonomy. We also recommend testing the quadcopter in extreme conditions such as strong winds or harsh weather in order to test its ability to filter interference and correct for error.

# Conclusion

Prof. Robert Dick, President, was contacted by a client who is planning to develop and market quadcopter platforms to enthusiasts and universities. We were asked by Prof. Dick to prepare a formal report about our quadcopter performance in the given maze as well as our custom project. We prepared this report to provide more details about our quadcopter performance and findings.

We were successful in our navigation of the maze, completing it in about 25 seconds with minimal collisions. Our main maze navigation algorithm was written in the C++ programming language. This code was uploaded to the quadcopter's main control system, which then routed control signals to the quadcopter. We controlled the quadcopter through Mission Planner, a ground control system for ArduPilot, which is a control software for drones and other autonomous systems. All code was written within a set autonomous controller, which was then modified through the use of code structures that incorporated selection.

Our custom design innovation is an autonomous landing algorithm based on color detection. This algorithm is intended to have real-world applications in the area of unmanned aerial vehicle safety. We completed the hardware setup of a camera for color detection, and also connected the camera to the quadcopter. Our goal was to mount the PixyCam setup on the quadcopter, fly it through the maze, and land whenever the quadcopter whenever it flew over a specific color. However, though we accomplished all other parts of this goal, we were unable to fly the quadcopter with the PixyCam setup due to time constraints and fragile hardware connections.

To further improve on our existing prototype for maze navigation, we recommend implementing a more autonomous method of maze navigation that will work for a variety of mazes. We also recommend further tuning PID control parameters for more efficient maze navigation. For our custom project involving the

PixyCam, we recommend incorporating algorithms such as edge detection to allow for features such as environment mapping. We were able to accomplish all the given tasks for the interested client, and have written this report to provide more details about our findings.

# References


[1] Cocco and Majenko, “Use all pins As digital I/O,” Arduino Stack Exchange, 21-Aug-2015. [Online]. Available: https://arduino.stackexchange.com/questions/14407/use-all-pins-as-digital-i-o. [Accessed: 10-Dec-2019].

[2] guych kawasaki. “CoDrone Tutorial: Throttle, Yaw, Pitch, Roll,” *Youtube,* May 2, 2018 [Video file]. Available: https://www.youtube.com/watch?v=FXabvMSQNxA. [Accessed: 10-Dec-2019].

[3] “PID controller,” Wikipedia, 03-Dec-2019. [Online]. Available: https://en.wikipedia.org/wiki/PID_controller. [Accessed: 10-Dec-2019].

[4] pixycam, “Pixy2, Pixy2 LEGO and Pan-tilt Quick Start,” Pixy Documentation, 18-Feb-2019. [Online]. Available: https://docs.pixycam.com/wiki/doku.php?id=wiki:v2:pixy_regular_quick_start. [Accessed: 10-Dec-2019].

# Appendix A: Maze Navigation Code

This appendix contains the code that we wrote within `control_autonomous.cpp`.

```
#include <iostream>
#include "Copter.h"

#define DIR_RIGHT 1
#define DIR_LEFT -1
#define DIR_BACKWARD 1
#define DIR_FORWARD -1


using namespace std;

/*
 * Init and run calls for autonomous flight mode (largely based off
of the AltHold flight mode)
 */

// autonomous_init - initialise autonomous controller
bool Copter::autonomous_init(bool ignore_checks)
{
    // initialize vertical speeds and leash lengths
    pos_control->set_speed_z(-g.pilot_velocity_z_max,
g.pilot_velocity_z_max);
    pos_control->set_accel_z(g.pilot_accel_z);

    // initialise position and desired velocity
    if (!pos_control->is_active_z()) {
        pos_control->set_alt_target_to_current_alt();

pos_control->set_desired_velocity_z(inertial_nav.get_velocity_z());
    }

    // stop takeoff if running
    takeoff_stop();

    // reset integrators for roll and pitch controllers
    g.pid_roll.reset_I();
    g.pid_pitch.reset_I();

    return true;
}

// autonomous_run - runs the autonomous controller
// should be called at 100hz or more
```

```
void Copter::autonomous_run()
{
    AltHoldModeState althold_state;
    float takeoff_climb_rate = 0.0f;

    // initialize vertical speeds and acceleration
    pos_control->set_speed_z(-g.pilot_velocity_z_max,
g.pilot_velocity_z_max);
    pos_control->set_accel_z(g.pilot_accel_z);

    // apply SIMPLE mode transform to pilot inputs
    update_simple_mode();

    // desired roll, pitch, and yaw_rate
     static float target_roll = 0.0f,  target_pitch = 0.0f,
target_yaw_rate = 0.0f;

    // get pilot desired climb rate
    float target_climb_rate =
get_pilot_desired_climb_rate(channel_throttle->get_control_in());
    target_climb_rate = constrain_float(target_climb_rate,
-g.pilot_velocity_z_max, g.pilot_velocity_z_max);

#if FRAME_CONFIG == HELI_FRAME
    // helicopters are held on the ground until rotor speed runup has
finished
    bool takeoff_triggered = (ap.land_complete && (target_climb_rate
> 0.0f) && motors->rotor_runup_complete());
#else
    bool takeoff_triggered = ap.land_complete && (target_climb_rate >
0.0f);
#endif
    target_climb_rate = 0.0f;

    // Alt Hold State Machine Determination
    if (!motors->armed() || !motors->get_interlock()) {
        althold_state = AltHold_MotorStopped;
    } else if (takeoff_state.running || takeoff_triggered) {
        althold_state = AltHold_Takeoff;
    } else if (!ap.auto_armed || ap.land_complete) {
        althold_state = AltHold_Landed;
    } else {
        althold_state = AltHold_Flying;
    }

    // Alt Hold State Machine
    switch (althold_state) {
```

```
    case AltHold_MotorStopped:


motors->set_desired_spool_state(AP_Motors::DESIRED_SHUT_DOWN);

attitude_control->input_euler_angle_roll_pitch_euler_rate_yaw(target_
roll, target_pitch, target_yaw_rate, get_smoothing_gain());
        attitude_control->reset_rate_controller_I_terms();
        attitude_control->set_yaw_target_to_current_heading();
#if FRAME_CONFIG == HELI_FRAME
        // force descent rate and call position controller

pos_control->set_alt_target_from_climb_rate(-abs(g.land_speed), G_Dt,
false);
        heli_flags.init_targets_on_arming=true;
#else
        pos_control->relax_alt_hold_controllers(0.0f);   // forces
throttle output to go to zero
#endif
        pos_control->update_z_controller();
        break;

    case AltHold_Takeoff:
#if FRAME_CONFIG == HELI_FRAME
        if (heli_flags.init_targets_on_arming) {
            heli_flags.init_targets_on_arming=false;
        }
#endif
        // set motors to full range

motors->set_desired_spool_state(AP_Motors::DESIRED_THROTTLE_UNLIMITED
);

        // initiate take-off
        if (!takeoff_state.running) {

takeoff_timer_start(constrain_float(g.pilot_takeoff_alt,0.0f,1000.0f)
);
            // indicate we are taking off
            set_land_complete(false);
            // clear i terms
            set_throttle_takeoff();
        }

        // get take-off adjusted pilot and takeoff climb rates
        takeoff_get_climb_rates(target_climb_rate,
takeoff_climb_rate);
```

```
        // get avoidance adjusted climb rate
        target_climb_rate =
get_avoidance_adjusted_climbrate(target_climb_rate);

        // call attitude controller

attitude_control->input_euler_angle_roll_pitch_euler_rate_yaw(target_
roll, target_pitch, target_yaw_rate, get_smoothing_gain());

        // call position controller

pos_control->set_alt_target_from_climb_rate_ff(target_climb_rate,
G_Dt, false);
        pos_control->add_takeoff_climb_rate(takeoff_climb_rate,
G_Dt);
        pos_control->update_z_controller();
        break;

    case AltHold_Landed:
        // set motors to spin-when-armed if throttle below deadzone,
otherwise full range (but motors will only spin at min throttle)
        if (target_climb_rate < 0.0f) {

motors->set_desired_spool_state(AP_Motors::DESIRED_SPIN_WHEN_ARMED);
        } else {

motors->set_desired_spool_state(AP_Motors::DESIRED_THROTTLE_UNLIMITED
);
        }

#if FRAME_CONFIG == HELI_FRAME
        if (heli_flags.init_targets_on_arming) {
            attitude_control->reset_rate_controller_I_terms();
            attitude_control->set_yaw_target_to_current_heading();
            if (motors->get_interlock()) {
                heli_flags.init_targets_on_arming=false;
            }
        }
#else
        attitude_control->reset_rate_controller_I_terms();
        attitude_control->set_yaw_target_to_current_heading();
#endif

attitude_control->input_euler_angle_roll_pitch_euler_rate_yaw(target_
roll, target_pitch, target_yaw_rate, get_smoothing_gain());
        pos_control->relax_alt_hold_controllers(0.0f);   // forces
throttle output to go to zero
        pos_control->update_z_controller();
```

```
        break;

    case AltHold_Flying:
        // compute the target climb rate, roll, pitch and yaw rate
        // land if autonomous_controller returns false
        if (!autonomous_controller(target_climb_rate, target_roll,
target_pitch, target_yaw_rate)) {
            // switch to land mode
            set_mode(LAND, MODE_REASON_MISSION_END);
            break;
        }


motors->set_desired_spool_state(AP_Motors::DESIRED_THROTTLE_UNLIMITED
);

#if AC_AVOID_ENABLED == ENABLED
        // apply avoidance
        avoid.adjust_roll_pitch(target_roll, target_pitch,
aparm.angle_max);
#endif

        // call attitude controller

attitude_control->input_euler_angle_roll_pitch_euler_rate_yaw(target_
roll, target_pitch, target_yaw_rate, get_smoothing_gain());

        // adjust climb rate using rangefinder
        if (rangefinder_alt_ok()) {
            // if rangefinder is ok, use surface tracking
            target_climb_rate =
get_surface_tracking_climb_rate(target_climb_rate,
pos_control->get_alt_target(), G_Dt);
        }

        // get avoidance adjusted climb rate
        target_climb_rate =
get_avoidance_adjusted_climbrate(target_climb_rate);

        // call position controller

pos_control->set_alt_target_from_climb_rate_ff(target_climb_rate,
G_Dt, false);
        pos_control->update_z_controller();
        break;
    }
}
```

```
// autonomous_controller - computes target climb rate, roll, pitch, 
and yaw rate for autonomous flight mode
// returns true to continue flying, and returns false to land
bool Copter::autonomous_controller(float &target_climb_rate, float 
&target_roll, float &target_pitch, float &target_yaw_rate)
{
      static int step = 1;
      //static int current_dir = 0;
      //static int current_hold = 0;


     // get downward facing sensor reading in meters
     float rangefinder_alt = (float)rangefinder_state.alt_cm / 100.0f;

     // get horizontal sensor readings in meters
     float dist_forward, dist_right, dist_backward, dist_left;
     g2.proximity.get_horizontal_distance(0, dist_forward);
     g2.proximity.get_horizontal_distance(90, dist_right);
     g2.proximity.get_horizontal_distance(180, dist_backward);
     g2.proximity.get_horizontal_distance(270, dist_left);

     // set desired climb rate in centimeters per second
     target_climb_rate = 0.0f;

      static int counter = 0;
      if (counter++ > 400) {
            // Send debugging messages: step we are on, target roll, 
and target pitch
            gcs_send_text_fmt(MAV_SEVERITY_INFO, "______________step: 
%d", step);
            gcs_send_text_fmt(MAV_SEVERITY_INFO, "roll: %f", 
target_roll);
            gcs_send_text_fmt(MAV_SEVERITY_INFO, "pitch: %f", 
target_pitch);

            // Reset counter to zero so that we send one message with 
the above info per second
            counter = 0;
      }


   // set desired yaw rate in centi-degrees per second (set to zero to 
hold constant heading)
      target_yaw_rate = 0.0f;


//*******************************************************************
*
```

```
      // Beginning of Modified Code

//***************************************************************
*

      // We run the default AltHold flight mode until we reach a
certain altitude, in this case, 25 cm

      // If we reach 25 cm altitude, then start running the
switch-case structure
      if (rangefinder_alt >= 0.25f) {

            // Start of switch-case
            switch (step) {

            case 1:

                  // For the first square of the maze, we read a
dist_left greater than 20 because the QC does not detect a wall
                  // If this is the case, we set a forward pitch so
that we move in our intended direction
                  if (dist_left > 20.0) {
                        g.pid_pitch.set_input_filter_all((5.0f -
dist_forward));
                        target_pitch = 10.0f * g.pid_pitch.get_pid();
                  }

                  // As soon as the QC sees a left wall, we maintain a
distance of 50 cm from that wall
                  if (dist_left <=20.0) {
                        // As soon as the QC sees a left wall, we
maintain a distance of 50 cm from that wall
                        g.pid_roll.set_input_filter_all((5.0f -
dist_left));
                        target_roll =  25.0*g.pid_roll.get_pid();

                        // We also continue to move forward using the
same forward PID filter as in the case where dist_left > 20.0
                        g.pid_pitch.set_input_filter_all((5.0f -
dist_forward));
                        target_pitch = 15.0f * g.pid_pitch.get_pid();
                  }

                  // When the QC encounters a wall (within 70 cm of the
wall), proceed to the next navigation step
                  if (dist_forward <= 7.0f) {
                        step = 2;
                  }
```

```
                // Break is here so that the code does not drop
through all the cases after this one
                break;

           case 2:
                // Make the QC move right until it sees a wall within
50 cm
                g.pid_roll.set_input_filter_all((5.0f - dist_right));
                target_roll = -10.0f * g.pid_roll.get_pid();

                // Make the QC hold the forward wall within 50 cm
                g.pid_pitch.set_input_filter_all((5.0f -
dist_forward));
                target_pitch =  10.0f * g.pid_pitch.get_pid();

                // If the QC sees a back wall (within 65 cm), proceed
to step 3
                if (dist_right <= 6.5f) {
                     step = 3;
                }

                // Break is here so that the code does not drop
through all the cases after this one
                break;

           case 3:

                // Make the QC move backward until it sees a wall
within 50 cm
                 g.pid_pitch.set_input_filter_all((5.0f -
dist_backward));
                 target_pitch = -15.0f * g.pid_pitch.get_pid();

                 // Make the QC hold the right wall within 50 cm
                 g.pid_roll.set_input_filter_all((5.0f - dist_right)
);
                 target_roll = -10.0f * g.pid_roll.get_pid();

                //Create a virtual wall so that we can use the same
step
                // We create this wall because there is not a right
wall all the way down this path
                if (dist_right >= 3.2f) {
                     dist_right = 3.0f;
                }
```

```
                // If the QC sees a back wall (within 80 cm), procees
to step 4
                if (dist_backward <= 8.0f) {
                     step = 4;

                }

                // Break is here so that the code does not drop
through all the cases after this one
                break;

           case 4:

                // Make the QC move right until it sees a wall within
50 cm
                 g.pid_roll.set_input_filter_all((5.0f - dist_right)
);
                 target_roll =  -15.0f * g.pid_roll.get_pid();

                 // Make the QC hold the back wall within 50 cm
                 g.pid_pitch.set_input_filter_all((5.0f -
dist_backward) );
                 target_pitch = -10.0f * g.pid_pitch.get_pid();

                // If the QC sees a right wall (within 50 cm),
proceed to step 5
                if (dist_right <= 5.0f) {
                     step = 5;
                }

                // Break is here so that the code does not drop
through all the cases after this one
                break;

           case 5:

                // Make the QC move forward until it sees a wall
within 50 cm
                g.pid_pitch.set_input_filter_all((5.0f -
dist_forward) );
                target_pitch =  15.0f * g.pid_pitch.get_pid();

                // Make the QC hold the right wall within 50 cm
                g.pid_roll.set_input_filter_all((5.0f - dist_right)
);
                target_roll = -10.0f * g.pid_roll.get_pid();
```

```
                // If the QC sees a forward wall (within 70 cm),
proceed to step 6
                if (dist_forward <= 7.0f) {
                     step = 6;
                }

                // Break is here so that the code does not drop
through all the cases after this one
                break;

           case 6:
                // Reset the step to 1 for the next run
                step = 1;
                // Lands the quadcopter
                return false;
           }

      }

      else  {

           // While we are still taking off and running default
AltHold
           // Reset the step to 1 default
           step = 1;
           // Set the target pitch to a default of 90 forward
(because the first square of the maze doesn't
           // have a left wall
           target_pitch = -90;
           // Set the default roll to zero
           target_roll = 0;

      }


      // Quadcopter keeps flying
      return true;


//*****************************************************************
*// End of Modified Code
//*****************************************************************
*
}
```

# Appendix B: PixyCam

This appendix contains the code that we wrote to connect the PixyCam to the Arduino and the Arduino to the BeagleBone.

```
/* VL53L1X_F19V1_1.ino simplified code for drone lidar heads
E100-400 F19 10/14/2019 V1.0
 * eng100-400 f19
 */

#define ALPHA 1.0f              // Tunable IIR filter on data to
mavlink

                                // Skips = Number of readings to skip
between printed outputs
                                // Skips = 0 will turn off printing
to console monitor
const uint8_t Skips = 30;

/*
*********  no user servicable parts below here  ***************
*/

// Libraries and headers

#include <Pixy2I2C.h>
#include <Pixy2.h>
#include <Wire.h>
#include <SoftwareSerial.h>
#include <VL53L1X.h>
#include "mavlink/common/mavlink.h"

Pixy2 pixy;


// Communication parameters for I2C and serial
#define BAUD_RATE 57600
#define CLOCK_HZ 400000

// Sensor configuration
#define N_SENSORS 4
#define XSHUT 4                           // XSHUT connects to
arduino pin 4
#define START_ADDRESS 42
#define DISTANCE_MODE VL53L1X::Medium     // range to 2.6 meters on
Medium
#define TIMEOUT_MS 500
```

```
#define TIME_BUDGET 30000                    // VL53L1X time budget in
microsec (20000 for short, 30000 for med, 50000 for long)
#define SAMPLE_TIME 36                       // sampling rate in
millisec - must be a few msec longer than budget

//---> Added by DG
#define LOOP_TIME 50
//<--- Added by DG

// MAVLink configurations
#define SYSTEM_ID 1
#define COMPONENT_ID 10
#define MIN_DISTANCE 0
#define MAX_DISTANCE 2600                     // Short: 1360, Medium:
2900, Long: 3600 in mm, change to match DISTANCE_MODE
#define COVARIANCE 0


// Global declarations
SoftwareSerial ardupilotSerial( 0, 1 ); // pins on the arduino. (rx,
tx)

VL53L1X sensor[N_SENSORS];                      // vl53l1x is a library
defined type, create an array of 4 of them
uint16_t distance[N_SENSORS] = {0};             // distance value set
uint32_t lastReadTime[N_SENSORS] = {0};         // time stamp array of
last good report for each sensor
float frequency = 0.0;                          // output sampling
frequency
uint16_t pcounter = 0;                          // print counter for
skipping outputs
boolean printFlag = true;                       // console serial
printer control flag

// Mavlink sensor orientation data array
uint8_t orientations[N_SENSORS] =
{
  MAV_SENSOR_ROTATION_NONE,           // forward-facing
  MAV_SENSOR_ROTATION_YAW_90,         // right-facing
  MAV_SENSOR_ROTATION_YAW_180,        // backward-facing
  MAV_SENSOR_ROTATION_YAW_270,        // left-facing
};


//FUNCTIONS

void send_distance(uint16_t dist, uint8_t channel )           // Send
ranging measurement to ot
```

```
{
  mavlink_message_t msg;
  mavlink_msg_distance_sensor_pack(
    SYSTEM_ID,
    COMPONENT_ID,
    &msg,
    (uint32_t)millis(),
    MIN_DISTANCE,
    MAX_DISTANCE,
    dist,
    MAV_DISTANCE_SENSOR_INFRARED,
    channel,
    orientations[channel],
    COVARIANCE
  );
  uint8_t buf[MAVLINK_MAX_PACKET_LEN];
  uint16_t len = mavlink_msg_to_send_buffer(buf, &msg);
  Serial.write(buf, len);
}

void printGoodData( uint8_t channel, uint16_t dist, float freq  )
{
  Serial.print( "Channel ");
  Serial.print( channel + 1 );
  Serial.print( " distance = " );
  Serial.print( dist );
  Serial.print(" mm, Freq = ");
  Serial.print( freq );
  Serial.print(" Hz");
}

void printBadData(String errMsg, uint8_t channel )
{
  Serial.print( "Channel ");
  Serial.print( channel + 1 );
  Serial.print( " " );
  Serial.print( errMsg );
  Serial.print( ", Reporting dist =  ");
  Serial.print( MAX_DISTANCE );
  Serial.print( " mm.");
}

void printFilteredDistance( uint16_t dist )
{
  Serial.print( ", filtered dist = " );
  Serial.println( dist );
}
```

```
void printAlpha()
{
  Serial.print("ALPHA is set to: ");
  Serial.print( ALPHA );
  Serial.println('\n');
}

static const uint8_t AddressDefault = 0b0101001;

//---> Added DG
void initializeI2CAddresses(uint8_t channel)
{
  digitalWrite( XSHUT + channel, LOW );
  delay( 10 );

  digitalWrite( XSHUT + channel, HIGH );
  delay( 10 );

  sensor[channel].setAddress(AddressDefault);
  sensor[channel].setTimeout( TIMEOUT_MS );                   // if a
sensor fails, punt
#if 0
if ( !sensor[channel].init() )
  {
    Serial.print( "Failed to initialize sensor " );
    Serial.print( channel + 1 );
    Serial.println();
    Serial.print( "Retrying..." );
    Serial.println();
    return;
  }
  #endif
  Serial.print("Connected!");
  Serial.println();
  sensor[channel].setAddress( START_ADDRESS + channel );      // Set
sensor's address and ranging mode
  sensor[channel].setDistanceMode( DISTANCE_MODE );
  sensor[channel].setMeasurementTimingBudget( TIME_BUDGET );
  sensor[channel].startContinuous( SAMPLE_TIME );
  Serial.print("Made it here\n");
}
//<--- Added DG


// Arduino Setup and Loop
void setup()
{
  pixy.init();
```

```
  if ( Skips ) Serial.begin( BAUD_RATE );                              //
Initialize UART and SW serial
                                                                      //
don't setup "Serial" if print is off
  ardupilotSerial.begin( BAUD_RATE );

  Wire.begin();                                                       //
Initialize I2C
  Wire.setClock( CLOCK_HZ );
/*
  //---> Modified by DG
  for ( uint8_t channel = 0; channel < N_SENSORS; channel++  )
pinMode( XSHUT + channel, OUTPUT );   // Disable all sensors
  for ( uint8_t channel = 0; channel < N_SENSORS; channel++  )
digitalWrite( XSHUT + channel, LOW ); // Reset
  for ( uint8_t channel = 0; channel < N_SENSORS; channel++  )
initializeI2CAddresses( channel );
  //<--- Modified by DG
*/

}

uint32_t lastlooptime_ms = 0;

void loop()                                     // read each sensor
once through the loop
{

  int i;
  pixy.ccc.getBlocks();


  // If the PixyCam detects Signature 1
  if (pixy.ccc.blocks[i].m_signature == 1){
      //Serial.println(pixy.ccc.blocks[i].m_signature);

      // Send distance to trick LIDAR into thinking it reads four 0's
      send_distance( 0, 0 );
      send_distance( 0, 1 );
      send_distance( 0, 2 );
      send_distance( 0, 3 );
    }

  //---> Added by DG
  delay(max(LOOP_TIME-lastlooptime_ms, 0));
  uint32_t time1 = millis();
  //<--- Added by DG
```

```
  if ( !Skips )
  {
    printFlag = false;                             // will never print
  }
  else
  {
    printFlag = !( Skips - ++pcounter );          // will print once
every Skips loops
  }
  if ( printFlag ) pcounter = 0;


  //---> Added by DG
  lastlooptime_ms = millis() - time1; // how much time this loop took
  //<--- Added by DG
}
```

# Appendix C: Technical Specifications

## PixyCam Specifications

To access our modified PixyCam code on Canvas: pixy_cam_stuff_final folder > pixy_cam_stuff.ino
The LIDAR initialization code is commented out in block comments. Our lines of modified code look like this:

```
// declare an integer variable to use in void loop()
int i;
// PixyCam native function to start the PixyCam reading
pixy.ccc.getBlocks();


  // If the PixyCam detects Signature 1
  if (pixy.ccc.blocks[i].m_signature == 1){

       // Debugging statement to confirm that the PixyCam is reading
       // Prints to the Arduino's serial monitor
       //Serial.println(pixy.ccc.blocks[i].m_signature);

       // Send distance to trick LIDAR into thinking it reads four 0's
      // Format : send_distance (distance, LIDAR channel)
       send_distance( 0, 0 );
       send_distance( 0, 1 );
       send_distance( 0, 2 );
       send_distance( 0, 3 );
    }
```

In addition to adding the code above, we also had to change some instances of `ardupilotSerial.begin` to `Serial.begin` because we were not able to send data from the PixyCam to the BeagleBone unless we did this. We are still not sure why this change to the code was necessary. We also had to configure the UART1 serial port that we opened in MissionPlanner. However, we were unable to find which parameters in MissionPlanner corresponded to the UART1 port, so we performed the UART1 serial port configuration on all serial ports.

## Details on Challenges and Challenge Solutions

Below are details about the challenges that we faced during different parts of this project and how we resolved these challenges.

**Maze Navigation Challenges**

- Frozen LIDAR sensors (2-3 days)

- Interference with LIDAR readings and printing too many (about 7) debugging messages from C++ code. Removed/decreased printing messages down to about 3 per function call
- Ultrasound readings read either 0 or over 7 meters (1 day)
    - Loose connection between BeagleBone board and sensor. Fixed this by resetting the board and reinforcing the connection between the board and sensor.
- Correct target pitch and roll values were not reading (about half the project time)
    - Variable scope problem. Needed to declare variables `static` since variables were being re-initialized every time function was called
- The `step` variable was not being reset to 1 when we landed (3-4 hours)
    - Added else statement at end and in step 6 that reset the `step` value to 1
    - This meant that the `step` variable would always be reset to 1 during the takeoff phase
- Quadcopter would not fly upward immediately (3-4 days)
    - Set a target altitude that contained the switch case function.
    - This meant that the code for the switch case would only be executed after the drone reached a certain altitude

**PixyCam Challenges**

- Setting up a second serial port (2-3 hours)
    - Needed connection between BeagleBone and PixyCam/second Arduino. John, one of the instructors found a way to open a second serial port and posted it so we could access it. However, we were required to build a voltage divider because of the voltage difference between the second Arduino board and beagleBeagleBone board
- Sketch would not upload to Arduino (1 hour)
    - The error in upload was due to pin 0 and 1 (the input and output pins) interfering with the serial data incoming from the BeagleBone and serial data going out from the computer.
    - This was fixed by removing the pins while uploading. This is known to affect only certain Arcuino boards such as the Uno, Due, and Mega.
- Modified LIDAR code was causing run issues on second Arduino (4-5 hours)
    - Code was mainly based off of given LIDAR code. When in the serial monitor, 'failed to initialize sensors' messages kept printing.
    - This meant the Arduino was thinking that it had LIDAR sensors (which it didn't) and was trying to initialize them
    - Fixed issue by commenting out any code that initialized the LIDAR sensors.
- Intended information (the PixyCam signature value and duped LIDAR values) were not sending to the BeagleBone (1 hour)
    - Caused by the `Serial.print` messages that we put into the code. We took them out and code functioned as intended.
- Could not find a way to send a MAVLINK message to tell drone to land (2 hours)
    - Found a way to"trick" the drone. Since drone uses LIDAR, we sent LIDAR values as 0 for all sides. Then, in the `autonmous_controller`, if all LIDARs read 0, we returned false
- Could not determine if second serial port was actually open and receiving/sending information as intended (2-3 days)
    - Used Linux environment to SSH into BeagleBone Blue root directory. Accecced the `/dev` file directory and opened the `ttyO1` file (corresponding to the UART1 serial port we opened).

    - Used Linux command `cat` which allowed us to see whether intended messages were being sent to the BeagleBone from the Arduino
- PixyCam was reading signatures even when signature was not present (verified through LED light) (1 hour)
    - Could be due to poor lighting conditions, lack of training, and/or poor choice of color.
    - Most likely, issue in software or hardware that caused faulty communication where pixycam sent a signal every time camera saw any signature rather than intended signature. Unable to find where in our system the problem was located